

# Verlässliche Echtzeitsysteme

## Redundante Ausführung

**Fabian Scheler**

Friedrich-Alexander-Universität Erlangen-Nürnberg  
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)  
www4.informatik.uni-erlangen.de

12. Juni 2012



## Gliederung

- 1 Überblick
- 2 Grundlagen
  - Fehlertypen
  - Redundanz
  - Replikation
  - Fehlerhypothese
- 3 Triple Modular Redundancy
- 4 Process-Level Redundancy
- 5 Diversität
- 6 Zusammenfassung



## Fragestellungen

- Maskierung **transienter Hardwarefehler** durch **redundante Ausführung**
  - **grundlegender Aufbau** replizierter Systeme
  - Auf **welcher Ebene** wird Redundanz angewandt?
  - Welche Eigenschaften müssen die einzelnen **Replikate** erfüllen?
- **Triple Modular Redundancy**
  - **die klassische Lösung** für die Auslegung fehlertoleranter Systeme
  - Replikation auf **Ebene des Knotens bzw. der Hardware**
  - Fokussierung auf **Triple Modular Redundancy**
    - prinzipiell sind n-fach redundante Systeme denkbar,  $n = 2, \dots$
- **Process-Level Redundancy**
  - redundante Ausführung unter Zuhilfenahme von **Mehrkernprozessoren**
  - Replikation auf **Ebene von Prozessen bzw. Software**
- Vermeidung von Gleichtaktfehlern durch **Diversität**
  - „replizierte Entwicklung“ der einzelnen Redundanzen



## Gliederung

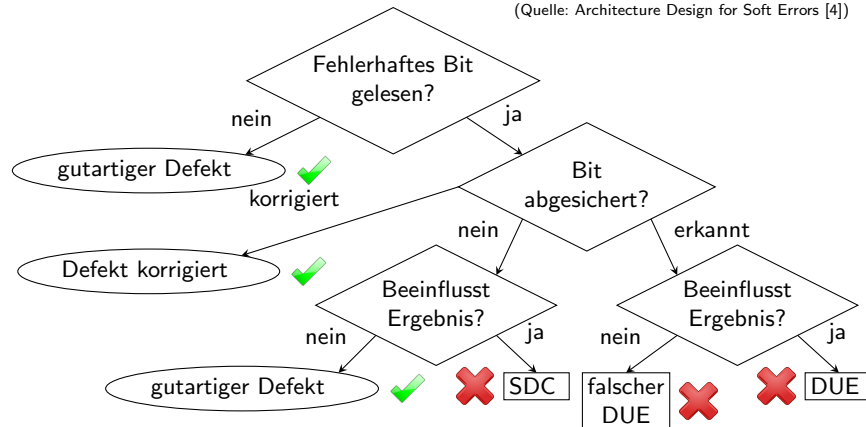
- 1 Überblick
- 2 Grundlagen
  - Fehlertypen
  - Redundanz
  - Replikation
  - Fehlerhypothese
- 3 Triple Modular Redundancy
- 4 Process-Level Redundancy
- 5 Diversität
- 6 Zusammenfassung



## Klassifikation – Fehlertypen

- Erinnerung: **transiente Fehler** (engl. *transient faults*) (s. Folie III/9)
  - treten wie sporadische Fehler **unregelmäßig** auf ...
  - münden i. d. R. aber nicht in einem permanenten Fehler
- **Datenfehler** (SDC) und **unkorrigierbare Fehler** (DUE, s. Folie III/11)

(Quelle: Architecture Design for Soft Errors [4])



## Fehlervermeidung

- **ungehemmte Fehlerfortpflanzung** führt zum Systemversagen
  - Datenfehler (engl. *silent data corruption*)
    - bedingen beispielsweise fehlerhafte Stellwerte für Aktoren
    - ihre Folgen treten häufig räumlich und zeitlich unkorreliert auf
  - erkannte, unkorrigierbare Fehler (engl. *detected unrecoverable errors*)
    - führen zu einem unmittelbaren, erkennbaren Systemversagen



Vermeidung dieser Fehler ist je nach Anwendung erforderlich

- **Problematic:** eine entsprechend robuste Auslegung einzelner Komponenten ist häufig nicht möglich
  - diese Komponente müsste frei von konzeptionellen Fehler sein, also keinerlei Hardware- oder Softwaredefekte etc. enthalten
  - sie müsste auch allerlei widrigen äußeren Umständen trotzen
- **Lösung:** man benötigt ein System, das einzelne Fehler tolerieren kann
  - einzelne Komponenten können ausfallen ...
  - dies wird durch andere **redundante Komponenten** aufgefangen,
  - die gewünschte Funktionalität an der Schnittstelle bleibt erhalten
  - der Anwender bekommt davon möglichst nichts mit (→ **Transparenz**)



## Fehlertoleranz durch Redundanz

- Fehlertoleranz erfordert immer **Redundanz**
- **räumliche Redundanz**
  - typisch für **hardwarebasierte** Fehlertoleranzlösungen
  - **mehrfache Auslegung:** Prozessoren, Speicher, Sensoren, Aktoren, ...
  - redundante Instanzen agieren häufig **simultan**

### zeitliche Redundanz

- typisch für **software-basierte** Herangehensweisen
- meist wird nur die Berechnung redundant ausgeführt
- **mehrfache, zeitlich versetzte Ausführung** redundanter Komponenten

### funktionale Redundanz

- mehrfache Herleitung desselben Sachverhalt auf verschiedenen Wegen
  - Geschwindigkeit ~ Drehzahlmesser bzw. Beschleunigungssensor
- orthogonal zu räumlicher und zeitlicher Redundanz

**Fokus:** zeitlich und räumlich redundante Ausführung

## Zustand redundanter Systeme

Wie verhalten sich die redundanten Systeme zueinander?

**hot standby** redundante Systeme arbeiten **simultan**

- sie verarbeiten gleichzeitig dieselben Eingaben
- ihr Zustand ist **jederzeit konsistent**
  - **nahtloser Ersatz** für ausgefallene Redundanzen

**warm standby** Unterscheidung von **Primär-** und **Sekundärsystem**

- Sekundärsystem läuft im **Hintergrund**
  - **regelmäßige Zustandssicherung** des Primärsystems (engl. *checkpoint*)
  - Rückkehr zur letzten Sicherung im Fehlerfall (engl. *recovery*)
- Primär- und Sekundärsystem sind zeitweise inkonsistent
  - höherer Aufwand im Falle der Fehlererholung

**cold standby** Sekundärsystem startet im Fehlerfall

- **unregelmäßige und eher seltene Zustandsicherung**
  - potentiell **großer Abstand der Redundanzen**
  - potentiell **langwierige Fehlererholung**

**Fokus:** redundante Systeme im „hot standby“-Betrieb

## Ziel der Redundanz

Was man mit dem Mehraufwand eigentlich bezweckt!

Fehlererkennung (engl. *fault detection*)

- Erkennen von Fehlern z. B. mithilfe von Prüfsummen

Fehlerdiagnose (engl. *fault diagnosis*)

- Identifikation der fehlerhaften redundanten Einheit

Fehlereindämmung (engl. *fault containment*)

- verhindern, dass sich ein Fehler über gewisse Grenzen ausbreitet

Fehlermaskierung (engl. *fault masking*)

- dynamische Korrektur von Fehlern z. B. durch Mehrheitsentscheid

Wiederaufsetzen (engl. *recovery*)

- Wiederherstellen eines funktionsfähigen Zustands nach Fehlern
  - Reparatur (engl. *repair*) bzw. Rekonfiguration (engl. *reconfiguration*)

Fokus: Fehlermaskierung

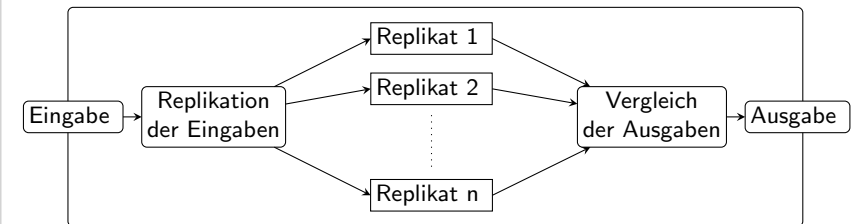
## Replikation

- Replikation ist der koordinierte Einsatz redundanter Elemente



Anordnung in einer „Sphere of Replication“ (SoR) [6]

- sie maskiert transparent Fehler in einzelnen Replikaten



- Eingaben werden repliziert und auf die Replikate verteilt
- in einem Ausgangsvergleich werden die Ausgaben abgestimmt
- Offene Fragestellungen:
  - Wie viele Replikate benötigt man, um das zuverlässig tun zu können?
  - Wie behebt man verbliebene kritische Bruchstellen?
    - Was passiert bei Fehler in der Eingabe oder im Ausgangsvergleich?

## Wie viele Replikate braucht man?

Unter der Annahme, dass höchstens  $f$  Fehler auftreten?

- Zahl benötigter Replikate hängt von der Art des Fehlverhaltens ab [3]
  - Annahme: von  $n$  Replikaten sind in folgender Weise  $f$  fehlerhaft

„fail-silent“

⇒ Anzahl der Replikate  $n = f + 1$

- ein Knoten erzeugt korrekt oder gar keine Antworten
- führt das Fehlverhalten zum Stillstand des Knoten  $\leadsto$  crash failure
- $\leadsto$  für einen Knoten der einfachste Fehlermodus
  - werden alle anderen Replikate informiert, ist das ein fail-stop failure

„fail-consistent“

⇒ Anzahl der Replikate  $n = 2f + 1$

- ein Knoten kann auch fehlerhafte Antworten erzeugen
- alle anderen Knoten sehen konsistent dasselbe Fehlverhalten

„malicious“

⇒ Anzahl der Replikate  $n = 3f + 1$

- „böartige“, fehlerhafte Knoten erzeugen verschiedene Antworten
  - die übrigen Knoten haben keine konsistente Sicht auf das Fehlverhalten
  - ggf. bekommt jedes Replikat eine andere (fehlerhafte?) Antwort
- Synonym: byzantinische Fehler (engl. *byzantine failures*)

## Wie viele Replikate braucht man? (Forts.)

- Vorabwissen kann helfen, die Zahl der Replikate zu reduzieren



hohe Fehlererkennungsrate (engl. *error detection coverage*)

- das Fehlverhalten wird innerhalb des Knotens erkannt
  - ein Ausbrechen des Fehler ist hier nicht tolerierbar
- umfasst sowohl Fehlverhalten im Wertebereich ...
  - falsche Eingabewerte oder Berechnungsergebnisse können beispielsweise durch Zusicherungen abgefangen werden
  - Durchführung häufig im Rahmen zyklischer Selbsttests
- als auch Fehlverhalten im Zeitbereich
  - einzelne Aufgaben überschreiten ihre maximale Ausführungszeit
  - „quasselnde Idioten“ (engl. *babbling idiot*) überlasten das Kommunikationssystem durch zeitlich unkoordinierten Nachrichtenversand
- $\leadsto$  das korrekte Systemverhalten ist a-priori bekannt
  - und kann genutzt werden, um „fail-silent“-Verhalten zu implementieren
  - zwei Replikate reichen in diesem Fall aus, um einen Fehler zu tolerieren



sonst: Mehrheitsentscheid liefert das korrekte Verhalten

$\leadsto$  hierfür benötigt man dann ein drittes Replikat

## Verbliebene kritische Bruchstellen

- **kritische Bruchstellen** (engl. *single points of failure*)
  - führen zu einem beobachtbaren Fehlerfall **innerhalb der Fehlerhypothese**
  - kompromittieren also die fehlertolerierende Eigenschaft der SoR
- ~ in der SoR auf Folie VII/10 sind dies **Eingabe und Ausgabe**



### Lösungsmöglichkeiten

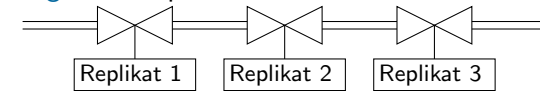
- bestimme Eingabedaten aus **mehreren Sensoren**
  - dies erfordert eine **Einigung der Replikate** über den Eingabewert, allen muss exakt derselbe Wert zugestellt werden
  - Anwendung funktionaler Redundanz ~ **Sensorfusion** (engl. *sensor fusion*)
- **repliziere den Ausgangsvergleich**
  - erneuter Mehrheitsentscheid über die Ergebnisse des replizierten Vergleichs
  - ~ das ist wieder eine kritische Bruchstelle, aber die **Fehlerwahrscheinlichkeit sind insgesamt geringer, verschwinden tut sie nie ...**
- **robuste Implementierung** des Ausgangsvergleichs
  - zusätzliche Absicherung des Ergebnisses durch z. B. **arithmetische Signaturen**
  - Durchführung des Mehrheitsentscheids durch den **Aktor**



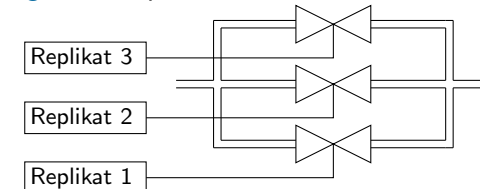
## Mehrheitsentscheid am Aktor

Am Beispiel von Rohrleitungen und Ventilen

- jedes Replikat kontrolliert jeweils ein Ventil
  - Vorgehensweise und Schaltfunktion ist hochgradig problemspezifisch
  - auch anwendbar auf elektronische Schaltkreise und Relais
- **Reihenschaltung** von Absperrventilen



- um den Fluss zu stoppen, genügt ein korrektes Replikat
- **Parallelschaltung** von Absperrventilen



- um den Fluss zu ermöglichen, genügt ein korrektes Replikat



## Fehlerisolation

- Replikate fallen **unabhängig** voneinander aus
  - **Gleichtaktfehler** (engl. *common mode failures*) sind zu vermeiden
    - sie führen zum **gleichzeitigen Ausfall mehrerer Replikate**
    - ~ eine Fehlermaskierung ist in diesem Fall nicht mehr möglich
  - **Quellen für Gleichtaktfehler** sind z. B. ...
    - **Softwaredefekte** und ...
    - das **Übergreifen eines Fehlers** auf andere Replikate



einzelne Replikate sind **gegeneinander abzuschotten**

- ein Dienst, den die SoR zur Verfügung stellt

### räumliche Isolation des internen Zustands

- dieser darf nicht durch andere Replikate korumpiert werden
  - ein verfälschter Zeiger hat großes Schadenspotential

### zeitliche Isolation anderer Aktivitätsträger

- eine Monopolisierung der CPU ist zu verhindern
  - ein Amok laufender Faden könnte in einer Schleife „festhängen“
  - selbiges gilt für alle gemeinsamen Betriebsmittel



## Lose Kopplung unterstützt Isolation

- Ziel sind **lose gekoppelte Replikate**
  - Minimierung des Koordinations- und Kommunikationsaufwands
    - je weniger sich einzelne Replikate abstimmen müssen, umso besser
- ~ Fehlerausbreitung wird auf diese Weise effektiv vermieden
- Unterstützung durch eine **statische, zyklische Ablaufstruktur**
  - 1 **Eingaben lesen**
    - der Zustand des kontrollierten Objekts wird erfasst
  - 2 **Berechnungen durchführen**
    - der neue Zustand wird aus dem alten Zustand und den Eingaben berechnet
  - 3 **Ausgaben schreiben**
    - die Stellwerte werden an die Aktoren ausgegeben
- lediglich die Schritte 1 und 3 erfordern eine Abstimmung der Replikate
  - Austausch von Nachrichten zwischen den Replikaten, um durch ein Einigungsprotokoll einen Konsens über die Eingaben/Ausgaben zu erzielen
- die Berechnung wird von jedem Replikat in „Eigenregie“ durchgeführt
  - ermöglicht einen **unterbrechungsfreien Durchlauf** (engl. *run-to-completion*)



## Replikdeterminismus

Korrekt arbeitende Replikate müssen identische Ergebnisse liefern.

- Replikate sind **replikdeterministisch** (engl. *replica determinate*), wenn
  - sie ihr von außen beobachtbarer Zustand identisch ist, und ...
  - sie zum ungefähr gleichen Zeitpunkt identische Ausgaben erzeugen
    - sie müssen innerhalb eines Zeitintervalls der Länge  $d$  erzeugt werden
    - im Bezug auf einen gemeinsamen Referenzzeitgeber
- Warum ist Replikdeterminismus wichtig?
  - Replikdeterminismus ist eine **Grundvoraussetzung für aktive Redundanz!**
  - korrekte Replikate könnten **unterschiedliche Ergebnisse** liefern
    - ein Mehrheitsentscheid ist in diesem Fall nicht mehr möglich
  - in den Replikaten kann **der interne Zustand divergieren**
    - unterschiedliche Ergebnisse sind die logische Folge
    - ein im Hintergrund laufendes Replikat kann im Fehlerfall nicht übernehmen
  - außerdem wird die **Testbarkeit** verbessert
    - schließlich kann man präzise Aussagen treffen, wann welche Ergebnisse von den einzelnen Replikaten geliefert werden müssten



## Phänomene, die Replikdeterminismus verhindern

abweichende Eingaben bei verschiedenen Replikaten

- **Digitalisierungsfehler**, z. B. bei der Analog-Digital-Wandlung
  - Temperatur- oder Drucksensoren liefern zunächst eine Spannung
  - diese Spannungen werden in einen diskreten Zahlenwert überführt
  - Abbildungen kontinuierlicher auf diskrete Werte sind fehlerbehaftet
- dies betrifft auch die **Diskretisierung der physikalischen Zeit**
  - ↪ **unterschiedliche Reihenfolge** beobachteter Ereignisse

unterschiedlicher zeitlicher Fortschritt der einzelnen Replikate

- Oszillatoren verschiedener Replikate sind nie exakt gleich
  - ↪ vor allem der Zugriff auf die lokale Uhr ist problematisch
  - u. U. werden **lokale Auszeiten** (engl. *time-outs*) deshalb gerissen

präemptive Ablaufplanung ereignisgesteuerter Arbeitsaufträge

- diese bearbeiten u. U. unterschiedliche interne Zustände
- die evtl. aus **Wetlaufsituation** (engl. *data races*) erwachsen sind

nicht-deterministische Konstrukte der Programmiersprache

- z. B. die **SELECT**-Anweisung der Programmiersprache Ada



## Wie stellt man Replikdeterminismus sicher?

globale diskrete Zeitbasis

- ermöglicht eine **globale zeitliche Ordnung** relevanter Ereignisse
  - ohne dass sich die Replikate hierfür explizit einigen müssen
- es dürfen **keine lokale Auszeiten** verwendet werden
  - betrifft die Anwendung, Kommunikations- und Betriebssystem

Einigung über die Eingabewerte

- die Replikate führen hierzu ein Einigungsprotokoll durch
  - konsistente Sicht bzgl. **Wert und Zeitpunkt** der Eingabe
- ↪ Grundlage für die globale zeitliche Ordnung aller Ereignisse

Statische Kontrollstruktur

- Kontrollentscheidungen sind **unabhängig von Eingabedaten**
  - ermöglicht außerdem eine statische Analyse dieser Entscheidungen
- Programmunterbrechungen sind mit größter Vorsicht einzusetzen

deterministische Algorithmen

- keine randomisierten Verfahren, nur stabile Sortierv Verfahren, ...



## Fehlerhypothese (engl. *fault hypothesis*)

Annahmen über das Verhalten einzelner Replikate im Fehlerfall

- in der Praxis betrachtet man für Echtzeitsysteme Replikate, die ...
  - **einen transienten Fehler** tolerieren können
  - sich „fail-silent“ oder zumindest „fail-consistent“ verhalten
  - **unabhängig voneinander** ausfallen
    - Gleichaktfehler müssen also ausgeschlossen werden
  - sich **replikdeterministisch** verhalten
    - ermöglicht eine einfache Umsetzung des Mehrheitsentscheids
- byzantinische Fehlertoleranz wird üblicherweise nicht angestrebt
  - Grund ist der **enorme Aufwand**, der damit verbunden ist
  - $3f + 1$  Replikate um  $f$  Fehler zu tolerieren
  - getrennte Kommunikationswege zwischen allen Replikaten
    - hoher Hardwareaufwand für Replikate und Verkablung
  - ↪ **hohe Kosten, Gewicht, Energieverbrauch**
  - Erkennung fehlerhafter Replikate erfordert aufwendige Kommunikation
    - $f + 1$  Kommunikationsrunden für  $3f + 1$  Replikate und  $f$  Fehler
    - je Runde schickt jedes Replikat eine Nachricht an alle anderen Replikate
  - ↪ für Echtzeitsysteme ein **nicht tolerierbarer zeitlicher Aufwand**



## Gliederung

### 1 Überblick

### 2 Grundlagen

- Fehlertypen
- Redundanz
- Replikation
- Fehlerhypothese

### 3 Triple Modular Redundancy

### 4 Process-Level Redundancy

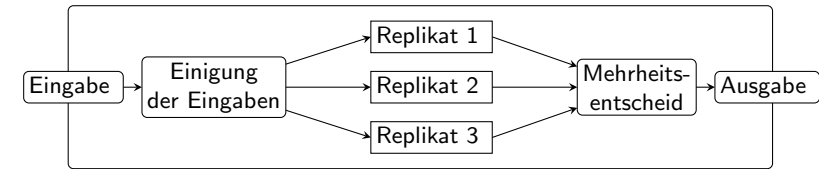
### 5 Diversität

### 6 Zusammenfassung



## Triple Modular Redundancy (TMR)

- falls Fehler im Wertebereich nicht zu verhindern sind



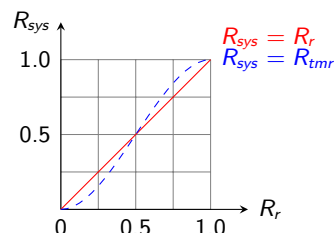
- üblicherweise dreifache Replikation kompletter Rechenknoten
  - räumlich redundante Systeme im „hot standby“-Betrieb
    - ~ weitgehende räumliche und zeitliche Isolation
- Abstimmung der Eingabewerte zwischen den Replikaten
  - die Replikate verfügen über eine gemeinsame globale Zeitbasis
  - das Kommunikationssystem verhindert die Steuerfehlerausbreitung
    - ~ vollständige zeitliche Isolation [5, Kapitel 8] und Replikdeterminismus
- Mehrheitsentscheid (engl. *voter*) stimmt Ausgabewerte ab
  - Vereinigung von Fehlermaskierung und -erkennung



## Wann hat TMR einen Nutzen?

Hilft viel grundsätzlich viel?

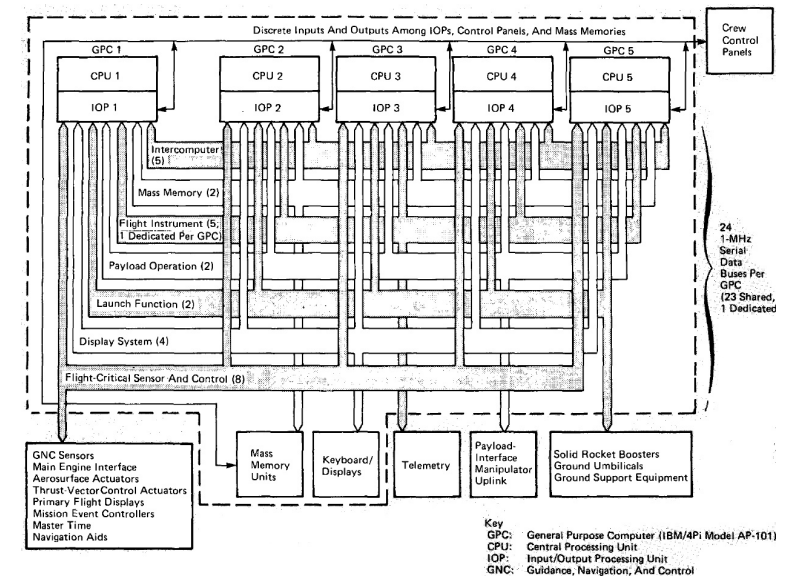
- Erhöht sich durch TMR in jedem Fall die Zuverlässigkeit?
    - anders formuliert:  $R_{tmr} > R_r$ ?
      - $R_{tmr}$  – Zuverlässigkeit des TMR-Verbunds,  $R_r$  des einzelnen Replikats
  - der TMR-Verbund arbeitet korrekt, solange ...
    - der Mehrheitsentscheid korrekt funktioniert  $\sim R_v$
    - zwei Replikate korrekt funktionieren  $\sim R_{2/3} = R_r^3 + 3R_r^2(1 - R_r)$ 
      - alle drei Replikate arbeiten korrekt oder ...
      - ein Replikat fällt aus, hierfür gibt es drei Möglichkeiten
- ~ insgesamt  $R_{tmr} = R_v(R_r^3 + 3R_r^2(1 - R_r)) > R_r$ ?



- Annahme: perfekter Voter  $R_v = 1$
- TMR ist nur sinnvoll falls  $R_r > 0.5$
- Praxis: Voter sollte zuverlässig sein
  - Größenordnung  $R_v > 0.9$



## Beispiel: Steuerung des Space Shuttle [2]

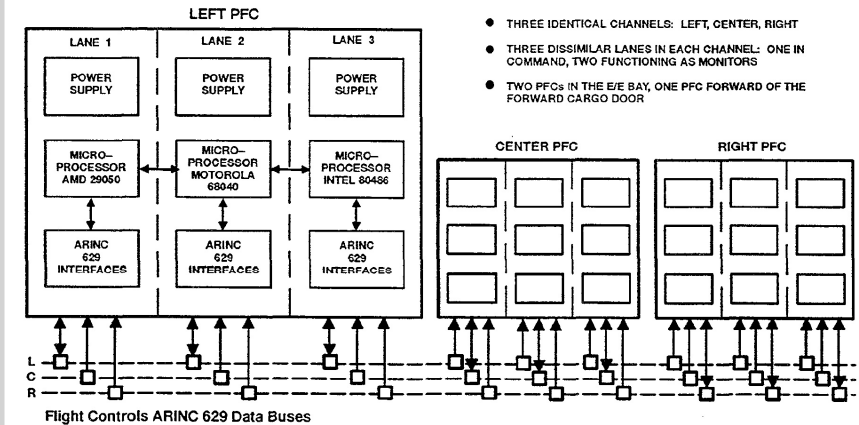




## Beispiel: Steuerung des Space Shuttle (Forts.)

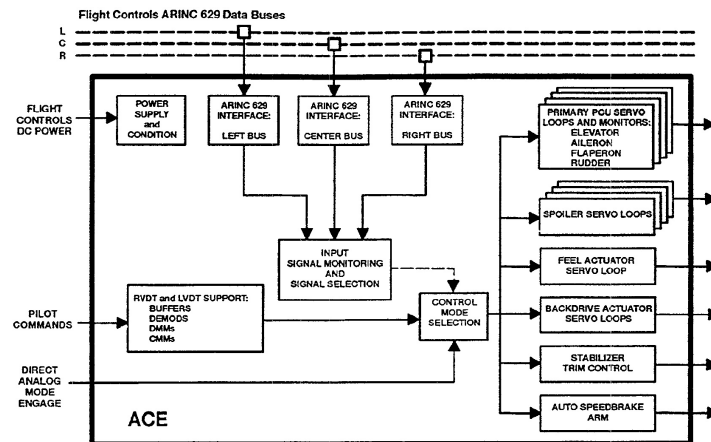
- insgesamt **fünf redundante Rechensysteme** [1, Kapitel 4.4]
  - ursprünglich gewünschte: **fail-operational/fail-operational/fail-safe**
    - Verlust eines Kontrollrechners ändert nichts an der Funktionsfähigkeit
    - das Gesamtsystem behält immer noch die Eigenschaft **fail-operational**
  - das war jedoch **zu teuer** ~ Reduktion auf vier Systeme
    - dies bedeutet **fail-operational/fail-safe**
    - das fünfte System war aber bereits überall eingeplant
    - ~ es wurde zu einem Backup-System „degradiert“ ~ „cold standby“
- unterschiedliche Konfiguration der Rechner je nach Missionsabschnitt
  - TMR nur im **Steigflug** bzw. **Sinkflug**
    - **drei Systeme** laufen simultan im „hot standby“-Betrieb
    - das vierte System läuft im „warm standby“
    - das fünfte System ist das Backup ~ „cold standby“
  - während des Shuttle in der **Umlaufbahn** ist, wird die Redundanz reduziert
    - **zwei System** laufen weiterhin simultan
    - das dritte System übernimmt Lebenserhaltungssysteme, ...
    - das vierte und fünfte Systeme sind Backup ~ „cold standby“

## Beispiel: Steuerung des Boeing 777 [7]



- **drei identische redundante Kanäle**: links, mitte, rechts
  - bestehend aus jeweils **drei diversitären redundanten Pfaden**
- **räumliche Verteilung** innerhalb des Flugzeugs
  - Minimierung der Auswirkungen z. B. von Blitzschlägen

## Beispiel: Steuerung des Boeing 777 [7] (Forts.)



- **Mehrheitsentscheid beim Aktor**
  - ACE = actuator control electronics
  - die Aktoren selbst sind ebenfalls redundant

## Gliederung

- 1 Überblick
- 2 Grundlagen
  - Fehlertypen
  - Redundanz
  - Replikation
  - Fehlerhypothese
- 3 Triple Modular Redundancy
- 4 Process-Level Redundancy
- 5 Diversität
- 6 Zusammenfassung

## Vorteile und Nachteile von TMR

### Vorteile von TMR

- sehr hohe Zuverlässigkeit bei richtigem Einsatz

### Nachteile von TMR

- enorm hoher Hardwareaufwand
  - ein Großteil der Hardwarekomponenten wird redundant ausgelegt
- hiermit direkt verbunden sind
  - hohe Kosten – viel Hardware kostet viel
  - hohes Gewicht – viel Hardware wiegt viel
  - hoher Energieverbrauch – viel Hardware benötigt viel Energie



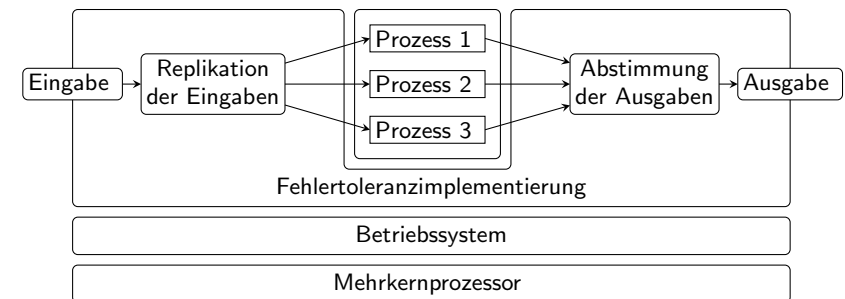
die höhere Integrationsdichte moderner Hardware könnte uns helfen

- auch wenn sie andererseits höhere Fehlerraten bedingt
- Mehrkernprozessoren „replizieren“ Rechenkerne
- sie erlauben die Ausführung mehrerer Replikate auf demselben Prozessor



## Process-Level Redundancy [6]

- Grundprinzip bleibt erhalten, nur der Inhalt der SoR ändert sich
  - es werden keine kompletten Rechenknoten mehr repliziert
  - sondern nur die Berechnung selbst, repräsentiert durch einen Prozess



- eine dedizierte Fehlertoleranzimplementierung sorgt für
  - die Replikation der Eingaben und die Abstimmung der Ausgaben
  - und die zeitliche Isolation der einzelnen Replikate
- hierfür greift sie auf ein Betriebssystem zurück
  - das räumliche Isolation sichert und Mehrkernprozessoren unterstützt



## Process-Level Redundancy [6] (Forts.)

- Funktionsweise der Fehlertoleranzimplementierung
  - Annahme: Replikate kommunizieren nach außen nur über Systemaufrufe
    - diese Annahme ist für Prozesse unter Linux durchaus valide



### Emulation der Systemaufrufchnittstelle

- lesende Systemaufrufe → Replikation der Eingabedaten
  - so findet automatisch eine Einigung über die Eingaben statt
- schreibende Systemaufrufe → Ausgaben puffern & Mehrheitsentscheid
  - nicht zurücknehmbare Seiteneffekte sind problematisch
  - sie dürfen erst durchgeführt werden, wenn ihre Korrektheit gesichert ist

### Synchronisation der einzelnen Replikate

- zu ähnlichen Zeitpunkten werden identische Systemaufrufe getätigt
  - sofern sich die einzelnen Replikate korrekt verhalten
- Überwachung durch Ausgangsvergleich und durch Auszeiten
  - die Fehlertoleranzimplementierung weiß, wann Systemaufrufe stattfinden

→ Replikdeterminismus

### zeitliche Isolation durch Überwachung der Laufzeit

- Überschreitung der Laufzeit führt z. B. zum Ablauf einer Auszeit



## Vergleich mit TMR

- Vorteil: Hardwareaufwand wurde deutlich reduziert
    - nur ein Prozessor (mit mehreren Rechenkernen)
    - kein gesondertes Kommunikationssystem zwischen den Replikaten
    - damit sind direkt verbunden
      - geringere Kosten, Gewicht, Energieverbrauch
  - Nachteil: der Grad an Redundanz nimmt unweigerlich ab
    - Fehler in gemeinsamen Teilen können zu Gleichtaktfehlern führen
      - Prozessorcaches, Stromversorgung, Kommunikationssystem
- Kompromiss aus Kosten und Nutzen

### Dennoch: Technologie der Zukunft

- Mehrkernprozessoren sind unaufhaltsam auf dem Vormarsch
  - erste dedizierte Mehrkernprozessoren im Automobilbereich
- gleichzeitig: einzelne Rechenkerne sind nicht mehr sicher genug
  - transiente Fehlerrate macht Redundanz unvermeidbar





## Gliederung

- 1 Überblick
- 2 Grundlagen
  - Fehlertypen
  - Redundanz
  - Replikation
  - Fehlerhypothese
- 3 Triple Modular Redundancy
- 4 Process-Level Redundancy
- 5 Diversität
- 6 Zusammenfassung



## Rückblick: Ariane 5 (s. Folie II/18)

- beide Inertialmesssysteme SRI1 und SRI2 fallen gleichzeitig aus
  - ein Ganzzahlüberlauf wegen einer Eingabe außerhalb der Spezifikation
  - ↪ die Bordcomputer OBC1 und OBC2 interpretieren den Fehlerwert falsch
  - ↪ fehlerhaftes Lenkmanöver führt zur Zerstörung der Rakete
- ☞ Ursache war ein **Gleichtaktfehler in homogenen Redundanzen**
  - Softwaredefekte sind typische Quellen für Gleichtaktfehler
  - Wie geht man mit Softwaredefekten um?
    - ↪ Wende **Redundanz bei der Entwicklung** solcher Systeme an!
- ☞ **Diversität** (engl. *diversity*) ↪ **heterogene Redundanzen**
  - auch **N-version programming**, mehr dazu siehe z. B. [3, Kapitel 6.6]
  - man nehme **mehrere verschiedene von allem**
    - Entwicklungsteams, Programmiersprachen, Übersetzer, Hardwareplattformen
    - alle entwickeln dasselbe System in mehreren Ausführungen
  - Annahme: die Ergebnisse sind für sich **wahrscheinlich nicht fehlerfrei**
    - ↪ aber sie enthalten **wahrscheinlich auch nicht dieselben Fehler**
    - ↪ Gleichtaktfehler dürften hier nicht mehr auftreten



## Diversität ist sehr umstritten!

- **Problem:** diese Annahme stimmt nicht unbedingt!
  - Gleichtaktfehler verursachende Defekte rühren oft aus der **Spezifikation**
  - ↪ diese betrifft alle diversitären Entwicklungsvorhaben gleichermaßen
    - was auch auf die Ariane 5 zugetragen hätte ...
- ☞ verwende **verschiedene Spezifikationen** als Ausgangspunkt
  - Wie bekommt man dann die „verschiedenen“ Ausgaben unter einen Hut?
  - dies erfordert **komplexe Verfahren** beim Mehrheitsentscheid
    - **exakte Mehrheitsentscheide** (engl. *exact voting*) sind vergleichsweise trivial
    - **unscharfe Mehrheitsentscheide** (engl. *non-exact voting*) sind aus heutiger Sicht hingegen nicht besonders vielversprechend ...
- Diversität findet dennoch erfolgreich Anwendung (s. Folie VII/26)
  - z. B. in asymmetrisch redundanten Systemen
    - eine komplexe Berechnung wird durch eine einfache Komponente kontrolliert
    - gepaart mit **fail-safe**-Verhalten im Fehlerfall
    - was bei Eisenbahnsignalanlagen sehr gut funktioniert
  - z. B. in der Reaktornotabschaltung vieler Kernkraftwerke
    - der Mehrheitsentscheid funktioniert nach dem Schema auf Folie VII/14



## Gliederung

- 1 Überblick
- 2 Grundlagen
  - Fehlertypen
  - Redundanz
  - Replikation
  - Fehlerhypothese
- 3 Triple Modular Redundancy
- 4 Process-Level Redundancy
- 5 Diversität
- 6 Zusammenfassung



## Zusammenfassung

Fehlertypen  $\mapsto$  Toleranz von SDCs und DUEs

Redundanz  $\mapsto$  hat mehrere Dimensionen

- Grundvoraussetzung für Fehlertoleranz
- räumlich, zeitlich, funktional, {hot, warm, cold} standby
- Fehlererkennung, -diagnose, -eindämmung, -maskierung

Replikation  $\mapsto$  koordinierter Einsatz von Redundanz

- Replikation der Eingaben, Abstimmung der Ausgaben
- Replikate für fail-silent, fail-consistent, malicious
- zeitliche und räumliche Isolation einzelner Replikate

Triple Modular Redundancy  $\mapsto$  Hardwareredundanz

- dreifache Auslegung, toleriert Fehler im Wertbereich
- Zuverlässigkeit von Replikat und Gesamtsystem

Process Level Redundancy  $\mapsto$  „TMR in Software“

- reduziert Kosten von TMR, zulasten eines geringeren Schutzes

Diversität  $\mapsto$  versucht Gleichtaktfehler auszuschließen



## Literaturverzeichnis

- [1] *Computers in Spaceflight: The NASA Experience.*  
<http://history.nasa.gov/computers/contents.html>, Apr. 1987
- [2] CARLOW, G. D.:  
Architecture of the space shuttle primary avionics software system.  
In: *Communications of the ACM* 27 (1984), Nr. 9, S. 926–936.  
<http://dx.doi.org/10.1145/358234.358258>. –  
DOI 10.1145/358234.358258. –  
ISSN 0001–0782
- [3] KOPETZ, H. :  
*Real-Time Systems: Design Principles for Distributed Embedded Applications.*  
Kluwer Academic Publishers, 1997. –  
ISBN 0–7923–9894–7
- [4] MUKHERJEE, S. :  
*Architecture Design for Soft Errors.*  
San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2008. –  
ISBN 978–0–12–369529–1
- [5] SCHELER, F. :  
*Echtzeitsysteme.*  
[http://www4.cs.fau.de/Lehre/WS11/V\\_EZS/](http://www4.cs.fau.de/Lehre/WS11/V_EZS/), 2011



## Literaturverzeichnis (Forts.)

- [6] SHYE, A. ; MOSELEY, T. ; REDDI, V. J. ; BLOMSTEDT, J. ; CONNORS, D. A.:  
Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance.  
In: *Proceedings of the 37th International Conference on Dependable Systems and Networks (DSN '07).*  
Washington, DC, USA : IEEE Computer Society Press, Jun. 2007. –  
ISBN 0–7695–2855–4, S. 297–306
- [7] YEH, Y. :  
Triple-triple redundant 777 primary flight computer.  
In: *Proceedings of the 1996 IEEE Aerospace Applications Conference.*  
Washington, DC, USA : IEEE Computer Society Press, Febr. 1996. –  
ISBN 978–0780331969, S. 293–307

